KARYA ILHAM PUBLISHING
CONNECTING MINDS, BRIDGING IDEAS, INSPIRING READERS

Journal of Advanced Research in Computing and Applications

Journal homepage:
https://karyailham.com.my/index.php/arca
ISSN: 2462-1927

Automated Unit Testing Practice Based on The Embedded Software Development Platform

Yingbei Niu[1,2,*], Soo See Chai[1], Kok Luong Goh[3], Kim On Chin[4]

[1] Faculty of Computer Science and Information Technology Universiti Malaysia Sarawak (UNIMAS) 94300 Kota Samarahan Sarawak, Malaysia
[2] Faculty of Education, Xi'an Jiaotong University, 710048 Xi'an, China
[3] Faculty of Computing and Software Engineering, i-CATS University College, Jalan Stampin Timur, 93350 Kuching, Sarawak, Malaysia
[4] Faculty of Computing & Informatics, University Malaysia Sabah, Kota Kinabalu, 88400, Malaysia

ARTICLE INFO

Article history:
Received 29 June 2025
Received in revised form 8 August 2025
Accepted 25 August 2025
Available online 4 September 2025

Keywords:
Automated test; test strategy; defect management; test coverage

ABSTRACT

Embedded software plays an essential role in modern technological systems, where quality and reliability are critical. However, traditional testing methods often face significant challenges in efficiency and coverage, creating a demand for more effective and comprehensive testing strategies. This study aims to explore automated testing within embedded software development, focusing on its advantages, methodologies, and impact on software quality. An experimental approach was adopted using automated unit testing tools to validate testing performance. The process covered functional, interface, user interface, and performance aspects. The results demonstrate that automated testing enables faster issue detection, improves precision, and enhances testing efficiency. Broad test coverage is achievable through well-structured automated unit testing, supported by best practices such as careful tool selection, clear and concise test scripting, early and continuous testing, and active stakeholder collaboration. Automated testing therefore offers a practical and efficient solution to improve software quality in embedded systems. Future research should examine the integration of automated and manual testing, security testing for embedded applications, and the application of machine learning and artificial intelligence to enhance testing capabilities.

1. Introduction

Embedded software is fundamental to modern technology and industry, underpinning systems ranging from smartphones to automotive controls and medical devices [1]. However, the escalating complexity of embedded software presents significant challenges for ensuring its quality and reliability, making effective software testing critically important [2]. Traditional manual testing methods often suffer from inefficiency, limited coverage, and difficulties in handling intricate scenarios such as endurance tests requiring uninterrupted long-duration execution or precise timed operations where manual precision is inadequate [3]. Consequently, automated testing has gained

* *Corresponding author.*
E-mail address: 2010175@siswa.unimas.my

https://doi.org/10.37934/arca.39.1.164180

prominence within embedded software development as a means to enhance testing efficiency, reduce manual effort, and ultimately ensure software quality. The integration of automated testing practices within embedded software development platforms offers a crucial solution, providing developers with precise and efficient testing capabilities [4,5]. Key advantages of automated testing include improved efficiency, particularly for special scenarios and repetitive tasks; increased test coverage enabling thorough examination of code; earlier defect detection lowering risks; and strengthened developer confidence in software quality [13-15].

This paper introduces and explores automated testing practices specifically founded on an embedded software development platform. We begin by examining the significance of automated testing in the embedded software domain and critically assessing the limitations inherent in traditional manual approaches. Subsequently, we elucidate the methodologies for automated testing enabled by the development platform, encompassing key phases such as test case creation, execution, and result analysis. Integrating these automated processes seamlessly into the development workflow allows for swift and efficient testing during iterative development cycles [6]. Automated testing is applicable across various types and levels within the software lifecycle. Depending on the phase and object, functional automation testing verifies stable core features [16], automatic interface testing focuses on component port requests and responses [17], UI automation testing assesses relatively stable graphical interfaces and workflows [18], and automatic performance testing handles tasks like daily scenario execution and anomaly analysis [19‑22]. At different test levels, unit testing automates verification of individual code units or components [23,24], configuration item testing examines the complete application flow including user interaction [25], and system testing validates overall user requirement fulfilment [25].

Prior research underscores the critical role of automated testing in assuring software quality and boosting development efficiency, particularly as embedded systems proliferate in safety-critical fields like aerospace and medical devices where stability and reliability are paramount [7,8]. Various approaches have been investigated, including platform-specific tool integration (e.g., using Testbed/Tbrun within the Tornado environment) for automating unit testing [9], comparative reviews of testing tools to aid selection [10], and model-driven techniques for automatic test case generation addressing embedded system needs [11]. However, existing studies often exhibit limitations. Some focus narrowly on specific tools or methods without providing a holistic view across multiple approaches [12], while others lack in-depth comparative analysis of manual versus automated testing effectiveness in practical, real-world embedded development scenarios [13].

Addressing these gaps, this paper focuses on the practical application of automated unit test case generation within the embedded platform context, as unit testing forms the bedrock of software verification. Automating the generation of high-coverage unit test cases with robust detection capabilities is essential for efficiently validating whether embedded software modules meet their expected functionality and logic [23,24]. Leveraging the tools and technologies inherent in the embedded software development platform, we investigate methods to achieve this automation effectively. Therefore, the central research question guiding this study is: How can efficient and effective automated testing, particularly unit testing across multi-modular code, be achieved on embedded platforms to enhance software quality and accelerate development?

The primary aim of this study is to explore the efficacy and viability of these platform-based automated testing practices. We validate these practices through concrete application, with a specific emphasis on the automated generation of unit test cases. Furthermore, we analyze the impact of such automation on software quality, testing efficiency, and development timelines within embedded software projects. Best practices for successful implementation, such as selecting appropriate tools, writing clear test cases, early and frequent automation, versioning support, and

incorporating user feedback [26-30], inform our approach. Given the relative scarcity of comprehensive research on automated testing integrated within embedded development platforms–especially concerning its multifaceted impacts throughout the development lifecycle–this work seeks to provide profound insights. The findings are expected to benefit embedded software developers by elevating testing quality and efficiency, reducing costs, expediting project delivery, and strengthening competitiveness. Ultimately, this research contributes to the broader goal of advancing technology to foster safer and more dependable embedded software products.

2. Methodology

An embedded software development platform is an information system with unified technical architecture, modularity, and integrated and distributed characteristics. Relying on demand management, the platform provides project management, test management, defect management, configuration management, pipeline management, RESEARCH and development efficiency and performance management, and basic support capabilities, and integrates and integrates professional construction, deployment, analysis, testing, and other tools, providing an effective support platform for software R & D management and control. In embedded software development, the automation test environment has the following important features and applicability:

i. Complexity of embedded systems: Embedded systems often contain a large amount of code and complex interactive relationships. With an automated test environment, one can automate large-scale test cases and capture possible errors and flaws. This helps improve test coverage and the ability to detect potential problems.
ii. Specific hardware and environment requirements: Embedded systems usually operate on specific hardware platforms and environments. The automated testing environment can simulate these hardware and environments and test for specific requirements. For example, sensor inputs, external interfaces, and real-time requirements can be simulated to ensure the stability and reliability of embedded code in an actual operating environment.
iii. Performance and security-oriented: Embedded systems often have high requirements for performance and security. The automated testing environment can evaluate the performance of embedded code in these aspects by simulating load, concurrency, and security attacks. This helps to identify and solve performance and safety issues in advance, reducing risks and costs later.
iv. Integration with the configuration management library: The integration of the automated test environment and the configuration management library can realize the automatic code extraction and update. After the developer submits the code, the environment can automatically extract the latest code from the configuration management library and execute the corresponding test cases. This integration performance ensures that the code used in the test environment is consistent with the code submitted by the developer, avoiding problems caused by version inconsistency.

Automated testing environment has important applicability in embedded code testing. It automates tests on the complexity of embedded systems, specific hardware and environment requirements, performance, and security, and integrates with the configuration management library to improve test efficiency and quality. Further research and practice can further explore and optimize the application of automated testing environments in embedded software development.

An automated testing framework based on an embedded software development platform can do the following:

The platform is associated with Klocwork, EvoSuite, and other automated test tools, which can choose one of them according to the needs, and then build an automated test environment suitable for embedded code based on the selected test tools. This environment can provide the settings and configurations necessary to ensure that the test tool can properly analyze and test the code.

The adaptability and precision of the code extraction from the configuration management library can be analyzed in-depth. First, the structure and organization of the configuration management library need to be considered to ensure that the required code segments can be accurately extracted. Second, it is necessary to ensure that the extracted version of the code is consistent with the version submitted by the developer, to avoid inconsistencies or errors. This can be achieved through the mechanism of versioning and code synchronization, ensuring that the environment has access to the latest code.

After the software developer completes the coding and submits it, the automatically extracts the corresponding code from the configuration management library and enters it into the automated test environment for testing. The key point here is the development of the predefined test rules. The predefined test rules should take into account the characteristics and requirements of the system-level code and the embedded code, including conditional branches, operators, judgment statements, etc. The rules should be complete, cover all aspects of possible problems, and can accurately detect potential defects.

The following can be considered when analyzing defect discovery capabilities and strengths of predefined test rules. First, the rules should be able to capture common coding errors and potential problems, such as null pointer references, boundary condition errors, etc. Second, the rules should be accurate and accurate to detect problems as early as possible stages to avoid more serious effects in the subsequent stages. Furthermore, the rules should be designed to incorporate the specific needs and domain knowledge of the project to ensure their applicability and validity in practical testing.

Through test management, details of use case execution and test reports can be viewed. This includes executed test cases, test results, defects found, etc. Testing management tools allow easy tracking and management of each link in the test process, providing the function of submitting defects to the project or iteration with one click. This can accelerate the speed of defect repair and improve the response efficiency of the development team.

The platform shall automatically record all automated test activities, test data, and test results for the coding quality of the statistical analysis software development users. This includes the generated test cases, coverage reports, defect information, etc. Through the analysis of these records, developers can evaluate the quality of coding, find common problem patterns and trends, and take corresponding measures to improve and optimize.

In conclusion, the adaptability and accuracy of the code extraction in the analysis from the configuration management database, as well as the integrity of the predefined test rules and the ability to find defects, is of great significance for establishing an effective automated test environment and improving test efficiency. At the same time, test management and statistical analysis can better manage the test process and evaluate the quality of coding, to improve the quality and efficiency of software development.

2.1 Automated Test Business Scenarios

The automated test based on the embedded software development platform calls the automated test service provided in the pipeline to directly submit and track defects, defect associated configuration versions; the test case script is automatically executed and calls the automated test service provided in the pipeline to automate the test of test components in the test environment [26,27], Figure 1 is the automated test business scenario.

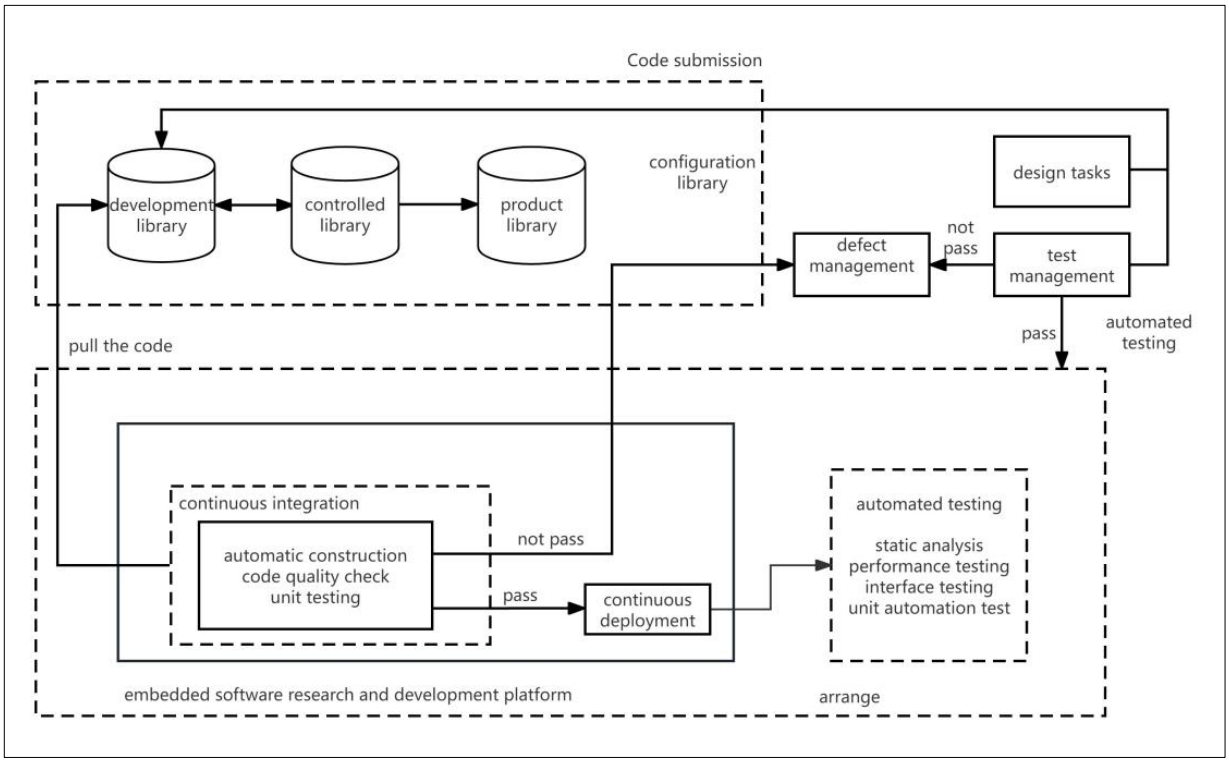


Fig. 1. Automatic test business scenario diagram

2.2 Automated Testing and Defect Management Process

Automatic test and defect management functions can cover the complete test management process, support the association between test cases and requirements and tasks, and form the association between test plans and iteration, forming a closed loop of the test process, improving test efficiency, and ensuring delivery quality. Figure 2 is a schematic diagram of the internal information relationship between automated testing and defect management:

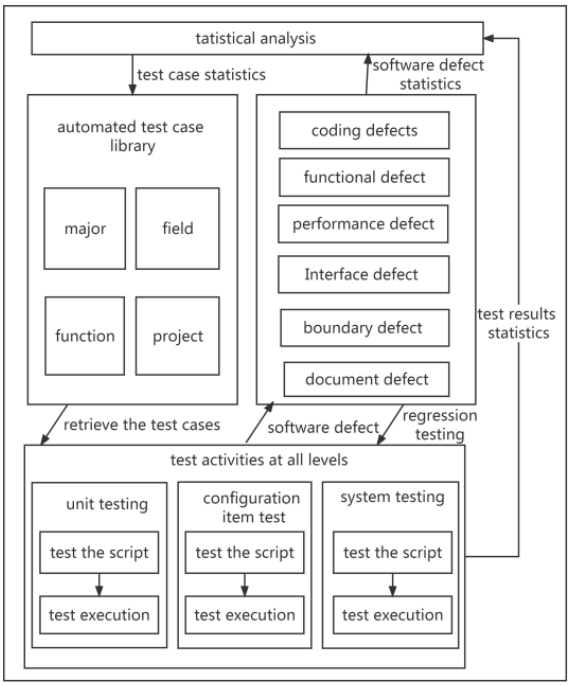


Fig. 2. Schematic diagram of the internal information relationship between automated test and defect management

The automated test case library can customize the use case attributes, adapt to different business scenarios, and support the use cases associated with product requirements and research and development tasks. The use case source and daily test process extracted from the test case library compiles the test scripts that have been successfully reused into the test case library and then reused by other projects. Test cases entered into the test case database will be related to the software requirements and software defects, which can easily be reused according to the software requirements, and can easily track and analyse software defects.

The software defects database automatically enters the software defects found in the automated test process after the test execution and conducts test data statistical analysis, including software defect classification and classification, and statistical analysis of test data to reflect the software coding quality and software product quality in real-time; the software defect report form is automatically generated after analysis, and the software defects can be assigned to corresponding developers for viewing, processing and forwarding. The test function matrix is shown in Table 1.

Table 1
Test the functional matrix

Order number	Functional steps	Task decomposition	Output products
1	Organize and manage the use case library	1) Build a use case library according to the requirements 2) Write test cases or select use cases in the use case library	Use case information
2	Organize and manage the defect library	1) Automatic collect defect results 2) Automatic analysis and display	Defective information
3	Automate test requirements analysis	1) Automatic test requirements analysis according to the software requirements 2) Establish a tracking matrix of test requirements and software requirements	Demand tracking matrix
4	Develop an automated test plan	1) Develop the test resource plan and schedule plan	Test plan, including the sequence of automated test execution for each function
5	Automated test case design and execution	1) Create the test cases 2) The platform automatically performs the test case script 3) Return the test results	Test cases and execution records
6	Generate the test report	1) The platform automatically arranges and analyzes the test data, evaluates the test effect and the tested software items 2) Generate software test reports and other related testing documents.	Test report, including defect diagnosis, code coverage results
7	regression testing	Impact domain analysis of defective functions or modules, and the test case is performed again after defect repair	Impact domain analysis report and regression test report

Based on the software requirements, analyze the automated test requirements, judge the test requirements, determine the content or quality characteristics of the automated test, and build the tracking matrix between the automated test requirements and the software requirements, so that the automated test requirements can be tracked to the corresponding software requirements.

Automatic test design is based on automated test requirements, designs test cases, and builds a tracking matrix between test cases and test requirements and software requirements. The coverage rate of test cases can reflect the adequacy of test design, which can be stratified and classified, reusable automated test cases can be modified, and the reuse, modification, and addition of test cases can be distinguished by identification [28-30]. Test case information elements include test case name, identification, test type, preconditions, test step, expected results, designer, design date, etc.

Automatic execution test cases, establish the tracking matrix of test case execution and software requirements, has passed the test case coverage of software requirements can reflect the technical status of the current software version, the software defects found automatically into the software defect database, and establish the tracking matrix of software defects and execution cases, facilitate the rectification of software defects to zero.

The regression test execution mainly includes the first round test, the first regression test, and the second regression test, involving the influence domain analysis, regression test case selection, and regression test data statistics. Test defect information elements include defect severity, defect distribution, test adequacy, etc.

3. Results

3.1 Manual and Automated Unit Testing Process and Comparative Analysis

As a means of checking and verifying the minimum testable unit, a unit test can find defects the first time and plays an important role in software quality assurance.

Due to the high writing cost of the unit tests, Is more needed to use automated generation in software testing, To write higher-quality test cases at a lower cost, The embedded software development platform integrates the EvoSuite unit test case automated generation tool, which can automatically generate test controls and complete, passable unit test scripts, simulate and encapsulate all function calls of the tested software, Provide optional automated detection parameters and data, test case call sequence validation, interface error detection and error injection, etc., Figure 3 is a schematic diagram of the test call control, Test case execution achieves 100% code coverage when checking the data, parameters, and call order.

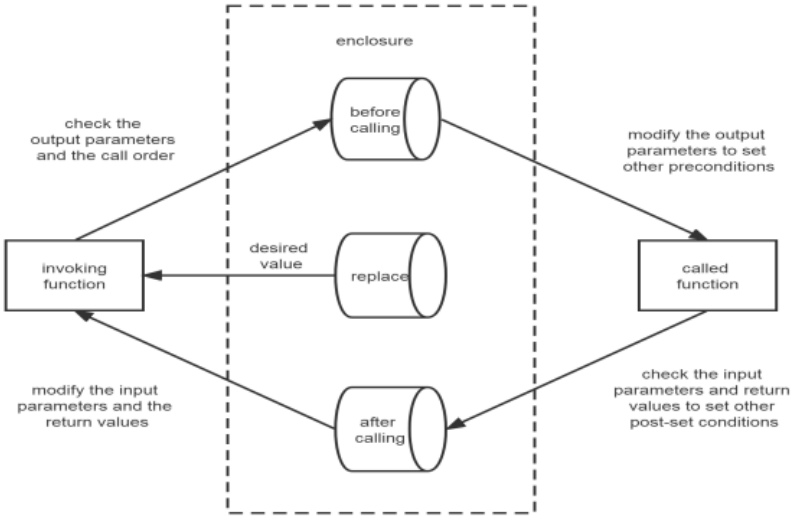


Fig. 3. Schematic diagram of the test call control

A function prototype in the header file can be used to generate test cases. Using tools to integrate automated testing improves the commonly used simple black box test for a complete white box test.

The automatically generated test script generates a test case for each function prototype defined in the header file. Create more test cases based on these use cases and avoid manually adding the information contained in the function prototype to the test case.

Test cases are associated at design time with the requirements being imported. The correlation between test cases, code, and requirements makes later code refactoring much easier.

Develop a smart home control system based on the JAVA language using MicroEJ on the embedded development management platform, The system can control lights, temperature, security cameras, and other devices. The system operates on an embedded device and requires a modular design to manage a variety of functions.

The embedded software contains 13,500 lines of code divided into 114 modules, each module responsible for different functions. The equipment control module is responsible for communication and control with various hardware devices (lights, temperature sensors, cameras, etc.). The communication module is responsible for handling communication with the user interface or external systems. Use the Java network programming library to establish communication with the

mobile application or remote-control interface to receive user commands and send instructions to the device control module. The user interface module is used to display the device status, receive user input, and provide user feedback. This can be done using Java's graph library. The security module is responsible for protecting the security of the system, possibly including authentication, access control, and data encryption. Log and debugging modules to facilitate troubleshooting and performance optimization, create a module to record logging and support remote debugging. The modular design can more easily manage the complexity of the embedded software and make the various parts relatively independent, thus improving the maintainability and scalability.

The following is the process of unit testing of one of the modules in both manual and automatic ways. The steps of manual unit test for the temperature acquisition and processing system of the developed based on MicroEJ on the embedded development management platform and automated unit test using EvoSuite are as follows:

Step 1: Environment Settings and project creation

In the Embedded Development Management Platform, open the New Project wizard, select the MicroEJ project type, and set the project name to MTMS and the target hardware platform to MicroEJ. Ensure that MicroEJ SDK is installed and configured in the development environment.

Step 2: Write the temperature acquisition and processing module code

Create the Java class TemperatureSensorModule and TemperatureProcessingModule in the project for the temperature sensor module and the temperature processing module, respectively.

Implement the corresponding functional methods in these classes, collectTemperature(), and convert to Fahrenheit (double Celsius).

Step 3: Write the manual unit test

Create a test directory test in the project to store the test code. In the test catalog, create test classes TemperatureSensorModuleTest and TemperatureProcessingModuleTest for the temperature sensor and processing module.

In the test class, test cases are written and then verified using the assertion. Use the command-line program to simulate manual testing. Create a TemperatureProcessing instance and call its method for temperature conversion and acquisition. Then, according to the actual output value, judge whether the test has passed. The part of code is as follows:

```
public class TemperatureProcessingManualTest {
   public static void main(String[] args) {
      TemperatureProcessing processor = new TemperatureProcessing();
      // Test temperature conversion
      double celsius = 20.0;
      double fahrenheit = processor.convertToFahrenheit(celsius);
      if (Math.abs(Fahrenheit - 68.0) < 0.01) {
         System.out.println("Temperature conversion test passed.");
      } else {
         System.out.println("Temperature conversion test failed.");
      }
```

```
        // Test temperature collection range
        double temperature = processor.collectTemperature();
        if (temperature >= -40.0 && temperature <= 125.0) {
            System.out.println("Temperature collection range test passed.");
        } else {
            System.out.println("Temperature collection range test failed.");
        }
    }
}
```

Step 4: Perform the manual unit test

Run the test code in the MicroEJ development environment. By right-clicking on the test class and selecting Run. Observe the output results. If the test passes, the output displays a success message; if the test fails, the output displays a failure message.

In study, both manual and automated testing phases utilized three distinct data sets, each containing a specific number of test cases. These data sets were meticulously curated to evaluate the robustness and efficiency of the testing methods. They are labeled as Data Set 1, Data Set 2, and Data Set 3:

Data Set 1: 342 test cases were completed in 28.5 hours, attaining 85% test coverage;
Data Set 2: Consisting of 350 test cases, it was wrapped up in 29 hours, achieving an 86% coverage;
Data Set 3: This set featured 346 test cases that were executed over 28.8 hours, securing 84% coverage.

To ensure thoroughness, results from each test case were systematically documented in an Excel spreadsheet. This rigorous record-keeping facilitated efficient monitoring of progress and swift identification of issues.

Step 5: Integrate the EvoSuite

Integrate the EvoSuite tool into the project, run EvoSuite, and let it analyze the code to generate test cases. Use the following command line:

```
evosuite -target .package.TemperatureProcessing
```

Step 6: Automatically generate the test cases

Using the EvoSuite tool, specify the classes to analyze (TemperatureSensorModule and TemperatureProcessingModule) and have it automatically generate test cases.

Open EvoSuite: Launch EvoSuite from the command line or use the integrated development environment (IDE) plugin if available. Ensure that the tool is properly configured to work with your project.

Specify Classes: In EvoSuite, specify the classes want to analyze for test case generation. In this case, mention TemperatureSensorModule and TemperatureProcessingModule as the target classes.

```
evosuite -class com.TemperatureSensorModule -class com.TemperatureProcessingModule
```

Generate Test Cases: Execute EvoSuite with the specified classes to trigger the test case generation process. EvoSuite will automatically analyze the classes and generate a set of test cases.

evosuite -class com.TemperatureSensorModule -class com.TemperatureProcessingModule -generateTests

Review the generated tests after EvoSuite completes the test case generation. The tests can be found in the output directory or as specified in the EvoSuite configuration. Ensure that the generated tests cover various scenarios and edge cases.

Save Generated Tests:Save the generated test cases in an appropriate directory within your project. These tests will be added to the MicroEJ development environment for verification.

Step 7: Run the automated unit test

EvoSuite generates a set of test cases that can be added to the MicroEJ development environment and run to verify the code.

The following are the test run results for the automated test phase:

Data Set 1: Comprised of 400 test cases, it was completed in 20 hours and achieved a coverage of 85%.
Data Set 2: This set contained 420 test cases, was executed in 21 hours, and reached a coverage of 92%.
Data Set 3: With 410 test cases, the tests took 20.5 hours and secured a 90% coverage.

It is noteworthy that the automated testing phase led to an improvement in test coverage by at least 5% when compared to the manual phase.The test time was saved by about 35%。 A comparative analysis of the results from both testing methods across the three data sets is illustrated in Figure 4.

Step 8: Analyze the test results

Analyze the test output generated by EvoSuite to confirm the coverage of test cases and test results.

The actual output of each test case was analyzed to check its agreement with the expected output. Ensure that the temperature module works properly under all conditions. When comparing the advantages of automated and manual testing in terms of coverage and efficiency, the following includes more accurate data descriptions.

A comparative analysis of the results from both testing methods across the three data sets is illustrated in Figure 4. As can be seen from Figure 4, the number of test cases for manual testing is lower in each dataset compared to automated testing. This indicates that manual testing requires fewer test cases to cover the same functionality. However, it is noteworthy that automated testing demonstrates significantly lower execution times (around 35%) compared to manual testing. This implies that automated testing is more efficient, as it can typically execute a large number of test cases in a shorter amount of time. In terms of test coverage, automated testing appears to perform better. The test coverage for automated testing is higher in each dataset compared to manual testing. This suggests that automated testing is more likely to achieve comprehensive test coverage. Overall,

automated testing shows advantages in efficiency and test coverage, as it can execute a greater number of test cases more quickly and achieve higher test coverage.

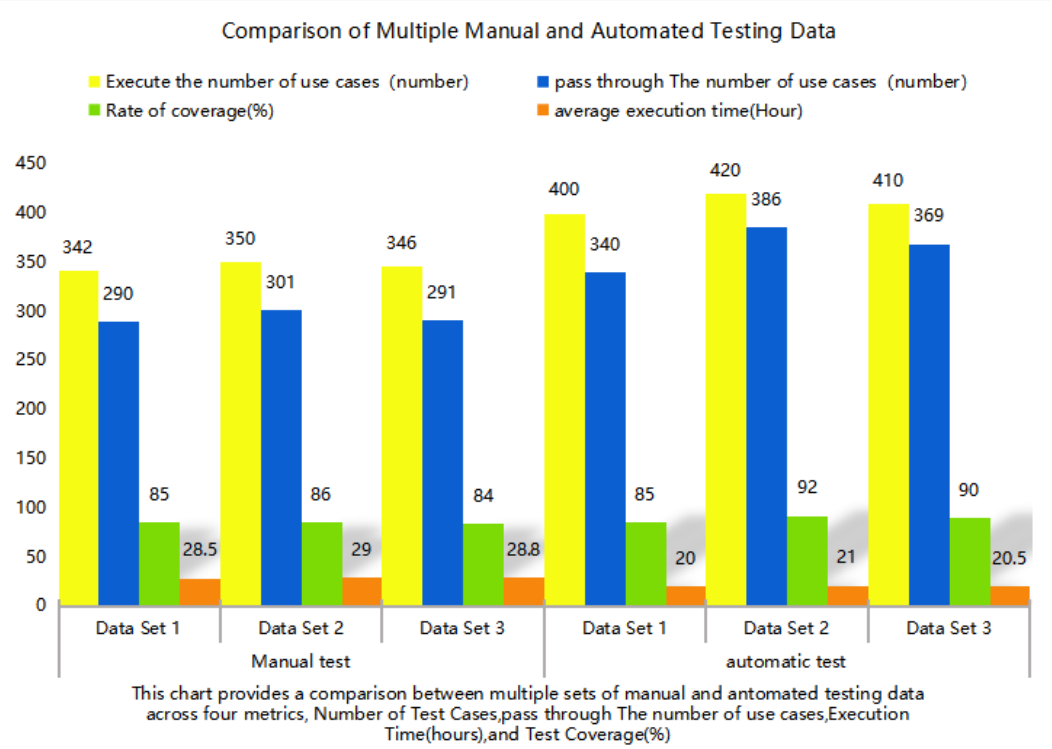


Fig. 4. Comparison of manual vs. automated testing results across three datasets

When considering test coverage, automated testing emerges as the superior option. Test coverage metrics consistently indicate that automated testing achieves higher coverage rates across all datasets compared to manual testing. This suggests that automated testing is better positioned to attain comprehensive test coverage. In summary, automated testing not only excels in efficiency but also outperforms manual testing in terms of test coverage. It can execute a larger number of test cases more swiftly while achieving broader test coverage. Moving forward, we will delve into a detailed analysis of the differences between manual and automated tests, focusing on test case density and defect density. In essence, automated testing presents three advantages:

It demonstrates high efficiency by rapidly processing a more significant number of test cases. It also maintains consistently broader test coverage. In terms of speed, it consistently reduces execution times, delivering results in nearly half the time.

When examining solely the test coverage metric, the superiority of automated testing becomes even more pronounced. Its consistent performance across datasets underscores its potential to deliver a more exhaustive examination of the software under test.

In conclusion, automated testing offers a compelling advantage, both in processing speed and in-depth coverage. Our next steps involve a granular analysis, delving into the nuances between manual and automated testing, with a particular emphasis on test case density and defect density.

3.2 Test Case Density Analysis

Test Case Density is a metric that quantifies the distribution of test cases in relation to a specific aspect of the software, often its size or complexity [31]. It is calculated as:

$$\text{Test Case Density} = \frac{\text{Number of Test Cases}}{\text{Software Metric}} \quad [1]$$

In the context of our study, the "Software Metric" in Equation [1] corresponds to the number of lines of code (LOC) in the software. From Table 2, it is evident that automated testing exhibits a consistently higher test case density relative to manual testing across all datasets. This suggests that automated testing likely provides more comprehensive test coverage in relation to the total lines of code within the software.

Table 2
Comparison of test case density between manual testing and automated testing

Test Type	Data set	Test Cases	Software Metric	Test Case Density (Test Cases per Line of Code)
Manual Testing	Set 1	342	13,500	0.0253
Manual Testing	Set 2	350	13,500	0.0259
Manual Testing	Set 3	346	13,500	0.0259
Automated Testing	Set 1	400	13,500	0.0296
Automated Testing	Set 2	420	13,500	0.0311
Automated Testing	Set 3	410	13,500	0.0304

3.3 Test Defect Density

Test Defect Density is a metric that measures the number of defects identified during testing in relation to a specific size or attribute of the software [32]. It provides insight into the quality of the software or the effectiveness of the testing process. It is calculated using the following formula:

$$\text{Test Defect Density} = \frac{\text{Average Defects Found}}{\text{Execution Time}} \quad [2]$$

Defect data from manual testing is meticulously recorded in an Excel sheet, featuring a comprehensive table that catalogs defect records for each testing dataset. This table encompasses essential details such as Defect ID, the number of test cases, test duration, test coverage, and the count of identified defects. As illustrated in Figure 5. When employing automated testing procedures, EvoSuite have the capability to generate comprehensive test reports once the test cases have been executed. These reports encompass detailed information about the outcomes of each test case, including any failures, along with additional statistical insights. Part of this test report section is presented in Figure 6（a）. View the automated test log as shown in Figure 6（b）.

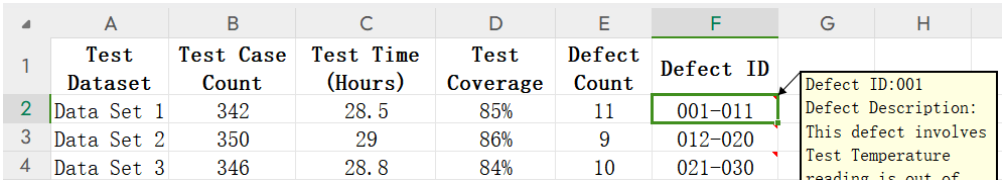
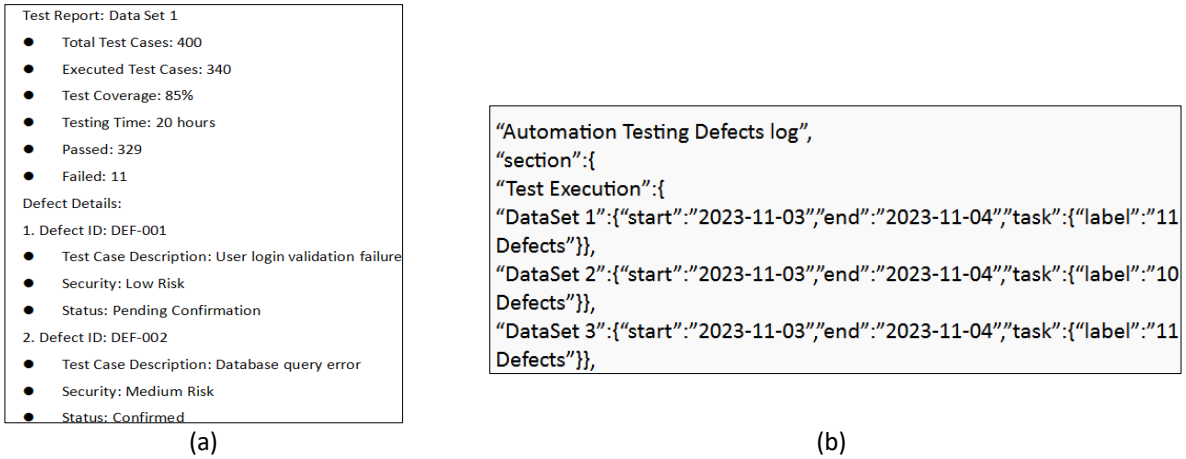


Fig. 5. Statistical table of the manual test data

Drawing upon the defect data analysis derived from the manual testing Excel records, it is evident that the average number of defects across the three datasets subjected to manual testing stands at 10.



(a) (b)

Fig. 6. Screenshots of (a) the partial test defect report and (b) the automated test defect log

Upon examination of the automated test reports, it becomes evident that the average number of defects across the three datasets subjected to automated testing is approximately 11.

From Table 3, manual testing's defect density across the datasets was approximately 0.38, 0.31, and 0.34 defects per hour. This indicates that roughly 0.31 to 0.38 defects are identified every hour during manual testing. A lower defect density here suggests a more stable software or fewer defects unearthed through manual means. Conversely, automated testing reported defect densities of 0.55, 0.47, and 0.53 defects per hour across the respective datasets. This means automated testing uncovers approximately 0.47 to 0.55 defects hourly. The elevated defect density with automated tests hints at their potential efficiency in defect detection. Nevertheless, the disparity might also relate to the volume of test cases and the test duration.

Table 3
Comparison of defect density between manual testing and automated testing

Test Type	Dataset	Defect Density (Defects per Hour)
Manual Testing	Set 1	0.38
Manual Testing	Set 2	0.31
Manual Testing	Set 3	0.34
Automated Testing	Set 1	0.55
Automated Testing	Set 2	0.47
Automated Testing	Set 3	0.53

In terms of test defect density, we could conclude that, automated testing often identifies more defects in a given time, resulting in a greater defect density. This could stem from its capability to execute more test cases swiftly and navigate through various code paths. Conversely, manual testing's lower defect density may signify more time taken to run identical test cases or a less expansive coverage, which consequently detects fewer defects.

4. Conclusions

It is very important to implement automated testing through applying an embedded software development platform. To achieve effective automated testing strategies, following best practices is key. This includes selecting the appropriate tools, writing clear and concise test case scripts, testing early and frequently, using versioning, and working with stakeholders. By following these best

practices, developers can ensure that their automated testing strategies are successful and that the quality of the software is improved.

The study encompassed two principal testing phases: manual and automated testing. During manual testing, three distinct datasets were evaluated, each demonstrating varying numbers of test cases, execution times, and test coverage percentages. Meticulous record-keeping of test results facilitated efficient progress tracking and issue identification throughout this phase.

In contrast, the automated testing phase employed an open-source test framework for test execution, resulting in significantly improved test coverage compared to manual testing. Automated testing not only achieved higher coverage rates but also demonstrated remarkable efficiency, executing a greater number of test cases in considerably less time.

When examining defect density, automated testing consistently yielded higher defect densities across different datasets, suggesting its effectiveness in defect detection. This could be attributed to its ability to efficiently execute numerous test cases and cover extensive code paths.

Moreover, a detailed analysis of the results revealed that automated testing not only detected the same defects as manual testing but also identified additional defects that were missed during the manual phase. This highlighted the superior comprehensiveness and efficiency of automated testing. Additionally, the automated approach significantly reduced execution times, achieving a 35% reduction compared to manual testing while concurrently enhancing test coverage.

Overall, the study underscores the advantages of automated testing, showcasing its efficiency and test coverage superiority over manual testing. It emphasizes the value of utilizing open-source tools and frameworks to enhance the testing of embedded software. The insights gained from both testing methods facilitated defect identification, software defect repairs, and performance optimization, ultimately contributing to the improved performance and responsiveness of the embedded software. Both testing methods proved effective in detecting system defects, but automated testing emerged as the preferred choice due to its efficiency and broader test coverage capabilities.

Looking ahead, future research can focus on advanced test automation techniques, including machine learning and AI, to boost efficiency and effectiveness. Integrating automated testing into CI/CD pipelines for rapid software delivery is a key area of exploration. Additionally, understanding how human testers can effectively collaborate with automated tools, using strategies like exploratory testing, is essential for comprehensive defect detection and efficient testing practices. These directions promise to enhance software testing in an evolving development landscape.

In summary, by adopting a variety of automated testing methods and following best practices, developers can establish effective automated testing strategies to ensure that the software is adequately tested and delivered on time. Future research directions will mainly include:

i. The deepening of software security testing is very important in the field of embedded software development. Future research will focus on deepening software security testing methods, including further development of static code analysis, dynamic analysis, and vulnerability scanning technologies. These methods will help automatically detect and fix security vulnerabilities to ensure the security and defense of embedded systems.
ii. The application of machine learning and artificial intelligence, and the application of machine learning and artificial intelligence technology will become an important trend of automated testing. Future research will explore how these techniques can be utilized to improve automated testing, including automated generation of test cases, intelligent defect detection, and automated analysis of test results. This will improve testing efficiency, accuracy, and adaptability, and help to better cope with complex embedded system testing requirements.

These research directions will drive the continuous development of automated testing technologies in the field of embedded software development to ensure software quality, safety, and reliability. At the same time, they can also help to improve testing efficiency, shorten development cycles, and adapt to the increasingly complex embedded system testing requirements.

Acknowledgement
The authors wish to thank Universiti Malaysia Sarawak for the financial support of this project.

References

[1] Durelli, Vinicius HS, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego RC Dias, and Marcelo P. Guimarães. "Machine learning applied to software testing: A systematic mapping study." *IEEE Transactions on Reliability* 68, no. 3 (2019): 1189-1212. https://doi.org/10.1109/TR.2019.2892517

[2] Swillus, Mark, and Andy Zaidman. "Sentiment overflow in the testing stack: Analyzing software testing posts on Stack Overflow." *Journal of Systems and Software* 205 (2023): 111804. https://doi.org/10.1016/j.jss.2023.111804

[3] Huong, Tran Nguyen, Lam Nguyen Tung, Hoang-Viet Tran, and Pham Ngoc Hung. "An automated stub method for unit testing c/c++ projects." In *2022 14th International Conference on Knowledge and Systems Engineering (KSE)*, pp. 1-6. IEEE, 2022. https://doi.org/10.1109/KSE56063.2022.9953784

[4] Gurcan, Fatih, Gonca Gokce Menekse Dalveren, Nergiz Ercil Cagiltay, Dumitru Roman, and Ahmet Soylu. "Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years." *IEEE Access* 10 (2022): 106093-106109. https://doi.org/10.1109/ACCESS.2022.3211949

[5] Al-Saqqa, Samar, Samer Sawalha, and Hiba AbdelNabi. "Agile software development: Methodologies and trends." *International Journal of Interactive Mobile Technologies* 14, no. 11 (2020). https://doi.org/10.3991/ijim.v14i11.13269

[6] Alakus, Talha Burak, Resul Das, and Ibrahim Turkoglu. "An overview of quality metrics used in estimating software faults." In *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pp. 1-6. IEEE, 2019. https://doi.org/10.1109/IDAP.2019.8875925

[7] Lu, Shuyi, Honghui Li, and Zhouxian Jiang. "Comparative study of open source software reliability assessment tools." In *2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS)*, pp. 49-55. IEEE, 2020. https://doi.org/10.1109/ICAIIS49377.2020.9194946

[8] Bui, Thu Anh, Lam Nguyen Tung, Hoang-Viet Tran, and Pham Ngoc Hung. "A method for automated test data generation for units using classes of qt framework in c++ projects." In *2022 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pp. 388-393. IEEE, 2022. https://doi.org/10.1109/RIVF55975.2022.10013869

[9] Alferidah, Saja Khalid, and Shakeel Ahmed. "Automated software testing tools." In *2020 International Conference on Computing and Information Technology (ICCIT-1441)*, pp. 1-4. IEEE, 2020. https://doi.org/10.1109/ICCIT-144147971.2020.9213735

[10] Tzoref-Brill, Rachel, Saurabh Sinha, Antonio Abu Nassar, Victoria Goldin, and Haim Kermany. "Tackletest: A tool for amplifying test generation via type-based combinatorial coverage." In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 444-455. IEEE Computer Society, 2022. https://doi.org/10.1109/ICST53961.2022.00050

[11] Steimle, Markus, Nico Weber, and Markus Maurer. "Toward generating sufficiently valid test case results: A method for systematically assigning test cases to test bench configurations in a scenario-based test approach for automated vehicles." *IEEE Access* 10 (2022): 6260-6285. https://doi.org/10.1109/ACCESS.2022.3141198

[12] Dvornik, Nikita, Julien Mairal, and Cordelia Schmid. "On the importance of visual context for data augmentation in scene understanding." *IEEE transactions on pattern analysis and machine intelligence* 43, no. 6 (2019): 2014-2028. https://doi.org/10.1109/TPAMI.2019.2961896

[13] Liang, Yuanzhen. "Application of Artificial Intelligence Algorithm and Deductive Database in Special Scene Recognition." In *2022 4th International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pp. 1012-1015. IEEE, 2022. https://doi.org/10.1109/ICSSIT53264.2022.9716323

[14] Chaudhary, Sonam, Ankur Choudhary, and Jyotsna Seth. "Nature Inspired Approaches for Test Case Selection in Regression Testing: A Review." In *2023 13th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 644-649. IEEE, 2023. https://doi.org/10.1109/Confluence56041.2023.10048797

[15] Choudhary, Bharat, and Shanu K. Rakesh. "An approach using agile method for software development." In *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, pp. 155-158. IEEE, 2016. https://doi.org/10.1109/ICICCS.2016.7542304

[16] Vijayasarathy, Leo R., and Charles W. Butler. "Choice of software development methodologies: Do organizational, project, and team characteristics matter?." *IEEE software* 33, no. 5 (2015): 86-94. https://doi.org/10.1109/MS.2015.26

[17] Elbanna, Amany, and Suprateek Sarker. "The risks of agile software development: learning from adopters." *IEEE Software* 33, no. 5 (2015): 72-79. https://doi.org/10.1109/MS.2015.150

[18] Ran, Dezhi, Hao Wang, Wenyu Wang, and Tao Xie. "Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing." In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 894-905. IEEE, 2023. https://doi.org/10.1109/ICSE48619.2023.00083

[19] Moghadam, Mahshid Helali, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. "Machine learning to guide performance testing: An autonomous test framework." In *2019 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pp. 164-167. IEEE, 2019. https://doi.org/10.1109/ICSTW.2019.00046

[20] Younas, Muhammad, Dayang NA Jawawi, Imran Ghani, Terrence Fries, and Rafaqut Kazmi. "Agile development in the cloud computing environment: A systematic review." *Information and Software Technology* 103 (2018): 142-158. https://doi.org/10.1016/j.infsof.2018.06.014

[21] Liao, Lizhi. "Addressing Performance Regressions in DevOps: Can We Escape from System Performance Testing?." In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 203-207. IEEE, 2023. https://doi.org/10.1109/ICSE-Companion58688.2023.00056

[22] Javed, Omar, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. "Perfci: A toolchain for automated performance testing during continuous integration of python projects." In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1344-1348. 2020. https://doi.org/10.1145/3324884.3415288

[23] Kumbhar, Saurabh, Prashant Bartakke, and Sanjay Kuchekar. "Software Test Automation of Electronic Control Unit." In *2022 International Conference on Recent Trends in Microelectronics, Automation, Computing and Communications Systems (ICMACC)*, pp. 1-5. IEEE, 2022. https://doi.org/10.1109/ICMACC54824.2022.10093382

[24] Bhaskar, Lavanya, Rahul B. Natak, and R. Ranjith. "Unit testing for USB module using google test framework." In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1-3. IEEE, 2020. https://doi.org/10.1109/ICCCNT49239.2020.9225528

[25] Zheng, Yu, Jun Shen, Ru Jia, and Ru Li. "Research on Automatic Generation of Comment Labels Oriented to Users' Individualized Needs." In *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 371-376. IEEE, 2022. https://doi.org/10.1109/CSCWD54268.2022.9776311

[26] Huang, Song, Sen Yang, Zhanwei Hui, Yongming Yao, Lele Chen, Jialuo Liu, and Qiang Chen. "Runtime-environment testing method for android applications." In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 534-535. IEEE, 2019. https://doi.org/10.1109/QRS-C.2019.00111

[27] Morales, Itza, Clifton Eduardo Clunie-Beaufond, and Miguel Vargas-Lombardo. "Coordination and Flexibility in the Management of Software Development Processes for Start-Up Companies." In *International Conference on Applied Technologies*, pp. 412-425. Cham: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-031-03884-6_30

[28] Dingsøyr, Torgeir, Finn Olav Bjørnson, Nils Brede Moe, Knut Rolland, and Eva Amdahl Seim. "Rethinking coordination in large-scale software development." In *Proceedings of the 11th international workshop on cooperative and human aspects of software engineering*, pp. 91-92. 2018. https://doi.org/10.1145/3195836.3195850

[29] Dirim, Sahin, and Hasan Sozer. "Prioritization of test cases with varying test costs and fault severities for certification testing." In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 386-391. IEEE, 2020. https://doi.org/10.1109/ICSTW50294.2020.00069

[30] Nyrud, Helga, and Viktoria Stray. "Inter-team coordination mechanisms in large-scale agile." In *Proceedings of the XP2017 scientific workshops*, pp. 1-6. 2017. https://doi.org/10.1145/3120459.3120476

[31] Kim, Jang-Eun, and Bo-Hyun Shim. "A study on Mass production stage Tank Battle Management System Environmental Stress Screening test method and application improvement based on Production process data." *Journal of Korean Society for Quality Management* 43, no. 3 (2015): 273-288. https://doi.org/10.7469/JKSQM.2015.43.3.273

[32] Nachiengmai, Wacharapong, Sakgasit Ramingwong, and Amphol Kongkeaw. "Implementing DDD for automatic test case generation." *Int. J. Inf. Educ. Technol.* 10, no. 2 (2020): 117-121. https://doi.org/10.18178/ijiet.2020.10.2.1349